Fig. 1

APPLICATION — 24

VIRTUAL MACHINE — 20

PROCESSOR — 30

MEMORY — 32

INTERFACE — 28

HARDWARE — 26

22

Fig. 1A

1000 (Forward Branch)

1002

OUTLIERS (STUBS)
- INFREQUENTLY EXECUTED

Fig. 1B

PROFILING COMPILING   FULL SPEED EXECUTING

NEW — 1020 (Invention)

OLD — 1022 (Prior Art)

10-SECS.

ADDRESS 1032
ENTRIES

4000	→ MAIN METHOD
5000	→ "METHOD" ("INVOKEGLUE")
6000	

1030
DISPATCH TABLE

FIG. 1C

```
JAVA CODE 1043
THRESHOLD TUNER 1056
DELETER 1062

CODE CACHE SEARCHER 1044
INTERPRETER 1042
EXECUTION HISTORY RECORDER 1050
COMPILER MANAGER 1052
MEMORY MANAGER 1060

CONVERTER DEVICE 1046
COMPILER 1058
QUEUE 1054

1040

EXECUTION DEVICE 1048
COMPILED CODE 1049
```

FIG. 1D

EXTERNAL
ENTRY
POINT
1074

1068 CODE

1070 CODE

1072 CODE

1066

1076 CONDITIONAL BRANCH

1088 DOMINANT PATH

A

1078 EXCEPTION HANDLER

B

1080 CODE

CALL/INVOKE

C

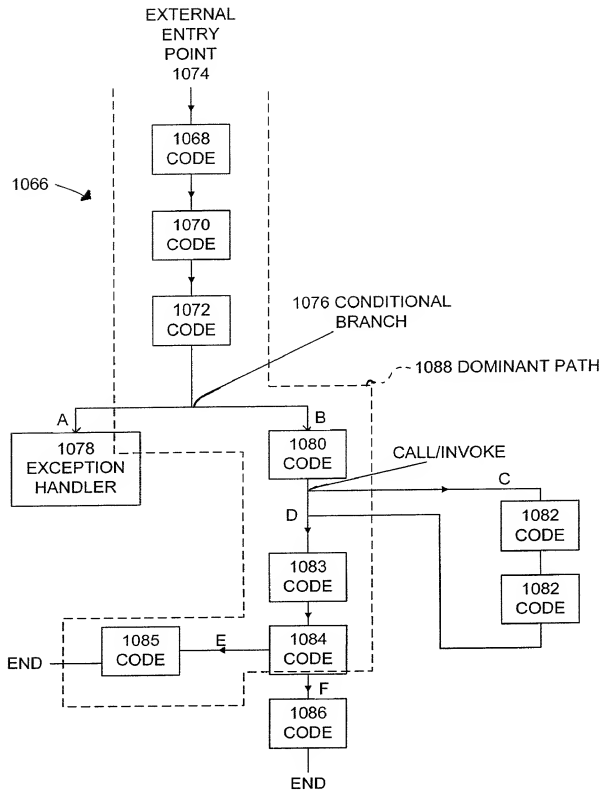1082 CODE

1082 CODE

D

1083 CODE

1084 CODE

E

1085 CODE

END

F

1086 CODE

END

FIG. 1E

FIG. 2A

FIG. 2B

FIG. 2C

FIG. 3A

FIG. 3B

FIG. 3C

FIG. 3A shows blocks B1, B2, B3, B4, B5 with labels:
- 9000, 9002, 9004 — EXCEPTION PATHS
- 9006, 9008 — DOMINANT PATH
- EXCEPTION CODE

FIG. 3B shows blocks B1, B2, B3, B4, B5.

FIG. 3C shows blocks B1, B3, B5, OL1, OL2.

COMPILED CODE

9054

B1

B2

B3

9056

CODE
BUFFER

9058

OL2   OL1   OUTLIERS

FIG. 3D

FIG. 3E

9206

PROCESSOR
CHIP

9209

9200   x86

400 MHz

FIRST LEVEL
16K CACHE

SECOND
LEVEL
512K
CACHE

9204

RAM
32M
<<400 MHz

9208

FIG. 3F

Fig. 4A

MODE OF TRANSPORT — 400

BIKE — 402

CAR — 404

- MODEL
- N° WHEELS

TURN ON LIGHTS

SALOON — 406

SPORTS — 408

Fig. 4B

412

414 — METHODS

ATTRIBUTES

START

410

Fig. 4C

OBJECT

HEADER | POINTER
DATA
DATA
DATA

420

DISPATCH TABLE

422

INDEX
426

INTERFACE HASH TABLE

424

Fig. 4D

Fig. 4F

Fig. 4E

Fig. 5A

Fig. 5B

Stack 29102

pointer to
previous frame
29106

29108
return pointer to
calling procedure's
executable code

Frames 29104

Fig. 5C

Stack 29102

frame pointer
return address

Store
29204

frame pointer
return address

special
code

Fig. 5D

29302 outstanding
procedure
call frame

procedure parameter 1
procedure parameter 2
⋮
procedure parameter n

29304

return address

29306

parent frame pointer

29308

local data

29310

Fig. 5E

procedure call stack 29402

pa1, pa2, pa3	← Frame for procedure A
return address	29412
A's parent frame
A's local data
pb1, pb2	← Frame for procedure B
return address (pt 1)	29410
B's parent frame
B's local data
pc1	← Frame for procedure C
return address (pt 2)	(youngest) 29408
C's parent frame
C's local data

29406

29404

Fig. 5F

barrier descriptor block 29502

barrier link
original return address
original parent frame
barrier function

Fig. 6A

~5008

NON-NATIVE CODE

a a a   call b b b

CALL INSTRUCTION

b b b   .

~5010   SUBROUTINE

return

RETURN ADDRESS

register
values

a a a   emulator
stack

~5009

Fig. 6B

Fig. 6C

Fig. 6D

DECIDE TO DELETE A CODE BUFFER ~5020

SELECT A THREAD TO EXAMINE ~5022

STOP THREAD ~5024

CREATE A RETURN BARRIER ~5026

SELECT A FRAME TO EXAMINE ~5028

5030 RETURN ADDRESS IN BUFFER ?

Y

FIND FRAGMENT WITHIN BUFFER THAT RETURN ADDRESS POINTS TO ~5032

ADJUST FRAME FIELDS TO "CLEAN UP" ~5034

CHANGE RETURN ADDRESS TO POINT TO GLUE CODE ~5036

GO TO NEXT YOUNGEST FRAME ~5040

N

LAST FRAME IN STACK ? ~5038

Y

START THREAD ~5042

LAST THREAD ? ~5044

N

Y

DELETE CODE BUFFER ~5046

Fig. 6E

Fig. 7A

FROM Space

TO Space

12108

inputs

live
objects
12106

most recent

12102   12104

Fig. 7B

data
structure

Heap

black/white
12210

growth

routes to other
objects
12212

12206

12202   12204

12208

Fig. 7C

black

white
d   12308

references

a

12306   b

c

12302   A

B   12304

Fig. 7D

STACK

ROOT SET
12404

ROOT
SET
12404

12400

12402

12402   12402

HEAP

12408

PACKETS 12406

Fig. 7E

12502 → | Header | M | ← 12504

object data

Fig. 7F

```
0
1
2
3
```

empty stack 12602

Fig. 7G

12406

packet link
max slots
occupied slots
slots

Fig. 7H

Fig. 8A

Sequence of Events for Various System Components *Fig. 8B*

Java Main Device Driver Thread	Java Interrupt Handling "Thread"	EVM Native Code	RTOS	Hardware
Requests and registers new int. handler "Thread"		Registers EVM entrypoint with RTOS int. handling services	→	
	Runs up to `waitForFirstInterrupt`	Kicks off new thread		
		Terminates thread but keeps stack and associates that stack with the relevant int.		
Issues I/O to device which will cause an int. to occur later	"In Limbo" (no RTOS thread... Just Java stack)			→
				(Later) Interrupt occurs
		Switch to limbo Java stack and do a return	RTOS calls our native code	
	Java int. handler code continues to run apparently in the original thread!			
	Call made on `waitForInterrupt`			
		Switch back to RTOS int. stack and return →		Interrupt Time-frame (as far as H/W is concerned)
			RTOS dismisses hardware int. (IRET) →	Interrupt is dismissed

Loop Forever

Fig. 8C-1

Pseudo-code of a Java Interrupt Handler

```
public void run ( )
{
        / / The run method of an example interrupt handling thread

        / / Wait for the first interrupt
        if  ( !waitForFirstInterrupt ( ) )
        {
                throw new RuntimeException  ( "Error waiting for 1st interrupt" ) ;
        }

        / / We are now running at interrupt level!

        while  (true)
        {
                / / Now handle the interrupt that just occurred (this involves
                / / reading a device register)

                byte value = dev.readByte (DEV.DATA_REG);

                if ( ( value & DEV.GOING_SYNC) != 0)
                {
                        / / We enter a sub-loop handling interrupts while in
                        / / "synchronous mode"

                        boolean stillSync = true;
                        do
                        {
                                / / Wait for the next interrupt to occur

                                waitForInterrupt ( ) ;

                                / / Read the hardware data register

                                value = dev.readByte  (DEV.DATA_REG) ;
```

go to Fig. 8C-2

Fig. 8C-2

from Fig. 8C-1

```
                        // Decide if the value means that we are switching back
                        // to "async mode"

                        if ( (value & DEV.GOING_ASYNC) == 0)
                        {
                                // Handle "synchronous mode" interrupt here (just
                                // write the device data to non-interrupt code via
                                // the special channel)

                                specialChannel.write (value);
                        }
                        else
                        {
                                // Wait for the next interrupt to occur then return
                                // to the outer "async" loop

                                waitForInterrupt ( ) ;
                                stillSync = false;
                        }
                }
                while (stillSync)
        }

        // Handle "asynchronous mode" interrupts here (just write the
        // device data to non-interrupt code via the special channel)

        specialChannel.write (value) ;

        waitForInterrupt ( ) ;
        }
}
```
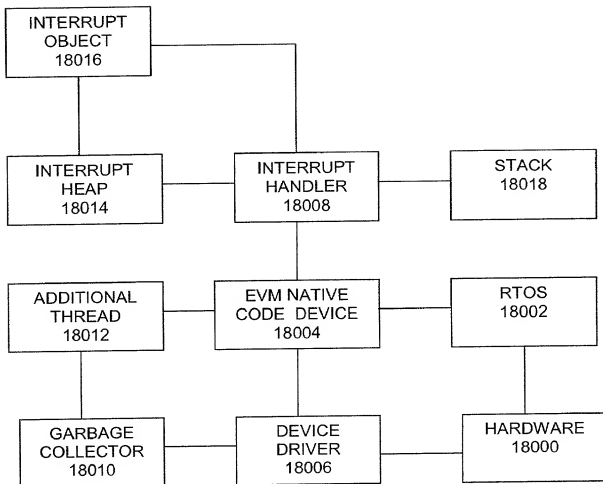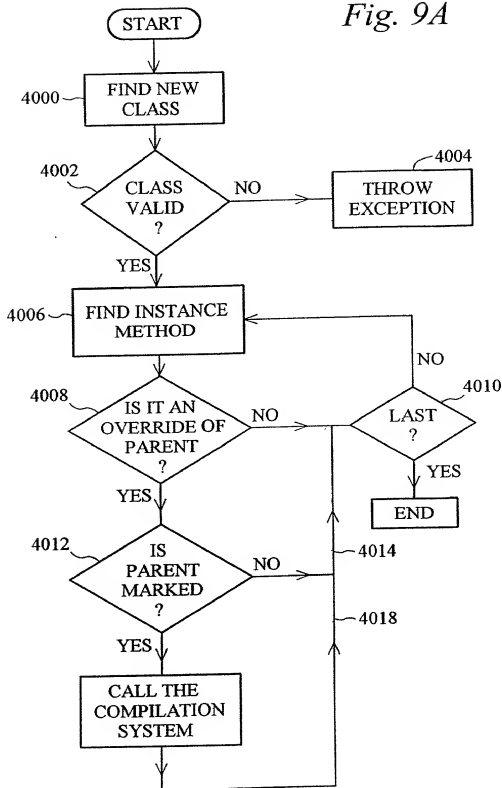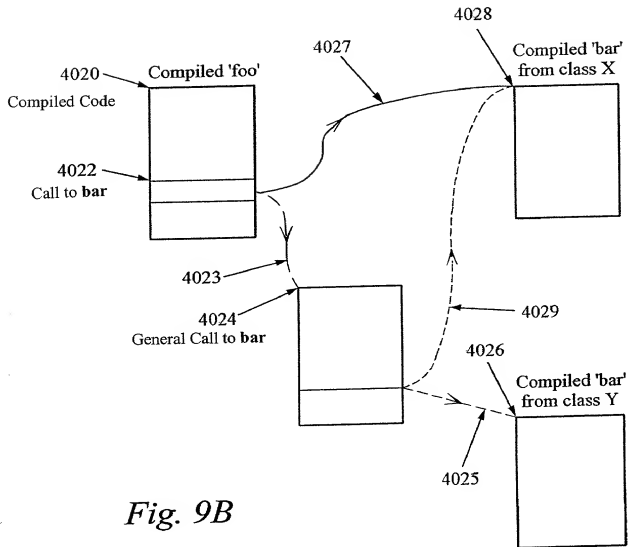
FIG. 8D

Fig. 9A

```
          ( START )

4000 ── FIND NEW
          CLASS

4002      CLASS        NO        4004
          VALID  ─────────────> THROW
          ?                      EXCEPTION

          YES

4006 ── FIND INSTANCE ──────────────────┐
          METHOD                         │
                                         │
4008      IS IT AN      NO        4010   │
          OVERRIDE OF ──────────> LAST   │
          PARENT                  ?      │
          ?                              │
                                  YES    │
          YES                            │
                                  END    │
4012      IS                             │
          PARENT        NO      ─4014    │
          MARKED ──────────────>         │
          ?                              │
                                ─4018    │
          YES                            │

          CALL THE                       │
          COMPILATION                    │
          SYSTEM                         │
```

Fig. 9B

Fig. 9C

FIG. 9D

Fig. 10A

LIST_HEAD
26104

NEXT_PTR | NEXT_PTR | NEXT_PTR | NULL
KEY | KEY | KEY | KEY
DATA | DATA | DATA | DATA
26110 | 26112 | | 26106

Fig. 10B

LIST_HEAD
26104

26202

NEXT_PTR | NEXT_PTR | NEXT_PTR | NEXT_PTR
KEY | KEY | KEY | KEY
DATA | DATA | DATA | DATA
26110 | 26112 | | 26106

Fig. 10C

26312

LIST_HEAD
26104

26202

NEXT_PTR | NEXT_PTR | NEXT_PTR | NEXT_PTR
KEY | KEY | KEY | KEY
DATA | DATA | DATA | DATA
26110 | 26112 | | 26106

26100
26102 | 26107 | 26108

MEMORY | PROCESSOR | STORAGE

Fig. 10D

Fig. 11A

code buffer 7102

code buffer

fragments 7100 of compiled code

Dymanic compiler state A

Dymanic compiler state A

code

JMP piglier

code

save off current contents of VM state elements not up to date but need to be

update VM state elements

call function to send SA

restore VM state contents

restore dynamic compiler state A

JMP back/return

'outliers' 7104 ('pigliers')

'piglier' 7106

Fig. 11B

No Pigging

Frag A } code before SP (B)

Dynamic compiler state R ⟶ • SP

} code after SP (C)

Pigging

Frag A } code before (B)

Dynamic compiler state R ⟶ JMP piglier

Dynamic compiler state R } code after (C)

Address   Instruction

23002   First Code Fragment
23003   Call Instruction

a a a   call abd

23004   Outlying Code
23005
a b d   call glue

Call Instruction   *Fig. 12A*

23002   Code Fragment
23003

a a a   call xyz

Patch
23008

x y z

23006
Compiled Code

return

23004   Outlying Code

Fig. 12B

a a a   call abc   — 23002
                     — 23003

x y z   — 23006
   return

a b c   jump x y z   — 23007   Jump Instruction
a b d   call glue   — 23004   Outlier

Fig. 12C

a a a   if - branch   — 23012   Section of Code
                        — 23013   Conditional Branch Instruction

x y z   — 23016

a b c   jump x y z
a b d   call glue   — 23014

Fig. 12D

Fig. 12E

DECIDE TO DELETE A CODE BUFFER — 23020

PRODUCE HASH OF BUFFER ADDRESS — 23022

PICK PATCH BLOCK OUT OF HASH CHAIN — 23024

23026 — PATCH TO BUFFER TO BE DELETED ? — N

23028 — REDIRECT PATCH TO GLUE CODE AND REMOVE PATCH BLOCK FROM CHAIN

23030 — LAST IN CHAIN ? — N

PICK PATCH FROM "FROM" (JUMP SOURCE) CHAIN — 23032

MAKE HASH OF "TO" BUFFER ADDRESS — 23034

REMOVE PATCH BLOCK FROM CHAIN — 23036

N — LAST IN CHAIN ? — Y — 23038

23039 — DELETE CODE BUFFER

Fig. 12F

NON-NATIVE CODE — 23042

23046

CONTROL

DELETE | INTERP-RETER | COMPILER | HASH TABLE | GLUE CODE | PATCH

23056 | 23048 | 23050 | 23052 | 23054 | 23055

23040
Virtual
Machine

MEMORY

PROCESSOR

Code Buffer
23062

23058

23060

23044

Fig. 12G